Software Interface Standards

Introduction

Over two thousand years ago, Carthage defined standard component sizes for their naval vessels, so that parts from one ship could be used to repair another. Slightly more recently, George Washington gave Eli Whitney a contract to build 12,000 muskets with interchangeable parts. The standardization of component interfaces makes it possible to successfully combine or recombine components made at different times by different people.

In the last half-century we have seen tremendous advances in both electronics and software, many of which are attributable to the availability of powerful, off-the-shelf components (integrated circuits, and software packages) that can be used to design and develop new products. We no longer have to reinvent the wheel for each new project, freeing us to focus our time and energy on the novel elements of each new design. As with ship and musket parts, the keys to this are interchangeability and interoperability: confidence that independently manufactured components will fit and work together. This confidence comes from the availability of and compliance with detailed component specifications.

In the early days of the computer industry, much key software (operating systems, compilers) was developed by hardware manufacturers, for whom compatibility was an anti-goal. If a customer's applications could only run on their platform, it would be very expensive/difficult for that customer to move to a competing platform. Later, with the rise of Independent Software Vendors (ISVs) and *killer applications*, the situation was reversed:

- The cost of building different versions of an application for different hardware platforms was very high, so the ISVs wanted to do as few ports as possible.
- Computer sales were driven by the applications they could support. If the ISVs did not port their applications to your platform, you would quickly lose all of your customers.
- Software Portability became a matter of life and death for both the hardware manufacturers and the software vendors.

If applications are to be readily ported to all platforms, all platforms must support the same services. This means that we need detailed specifications for all services, and comprehensive compliance testing to ensure the correctness and compatability of those implementations.

The other big change was in who the customers for computers and software were. In the 1960s, most computers were used in business and technology, and the people who used computers were professional programmers: trained and prepared to deal with the adverse side effects of new software releases. Today, most computers are in tablets, phones, cars, televisions, and appliances; and the people who use them are ordinary consumers who expect them to just work. These changes imply that:

- New applications have to work on both old and new devices.
- Software upgrades (e.g. to the underlying operating system) cannot break existing applications.

But the number of available applications, and the independent development and delivery of OS upgrades and new applications make it impossible to test every application with every OS version (or vice versa). If the combinatorics of the problem render complete testing impossible, we have to find another way to ensure that whatever combination of software components wind up on a particular device will work together. And again, the answer is detailed specifications and comprehensive compliance testing.

The last forty years have seen a tremendous growth in software standards organization participation (both by technology providers and consumers) and the the scope, number, and quality of standards.

Challenges of Software Interface Standardization

When a technology or service achieves wide penetration, it becomes important to establish standards. Television would never have happened if each broadcaster and manufacturer had chosen different formats for TV signals. In the short term, a slightly better implementation (e.g. cleaner picture) may give some providers an advantage in the market place. But ultimately the fragmentation of the marketplace ill-serves both producers and consumers.

If the stake-holders get together and agree on a single standard format for TV broadcasts, every TV set can receive every station. Consumers are freed to choose any TV set they want to watch any stations they want. Since all TV sets receive the same signals, set manufacturers must compete on the basis of price, quality, and other features. The cost of TV sets goes down and their quality goes up. Since all stations can reach all viewers, they must compete on the basis of programming. The amount, quality, and variety of programming increases. And the growing market creates opportunities for new products and services that were previously unimaginable.

Standards are a good thing:

- The details are debated by many experts over long periods of time. The extensive review means they will probably be well considered, and broad enough to encompass a wide range of applications.
- Because the contributors work for many different organizations, with different strengths, it is likely that the standards will be relatively *platform-neutral*, not greatly favoring or disadvantaging any particular providers.
- Because they must serve as a basis for compatible implementations, they tend to have very clear and complete specifications and well developed conformance testing procedures.
- They give technology suppliers considerable freedom to explore alternative implementations (to improve reliability, performance, portability, maintainability), as long as they maintain the specified external behavior. Any implementation that conforms to the standard should work with all existing and future clients.

But standardization is the opposite of diversity and evolution, and so is a two-edged sword:

- At the same time, they greatly constrain the range of possible implementations. An interface will specify who provides what information, in what form, at what time. It might be possible to provide a much better implementation of that functionality, if slightly different information could be provided at a different time. But if the new interfaces are not 100% upwards compatible with the old ones, the improved design would be non-compliant. Television was pioneered in the United States, but because we were among the first to standardize (e.g. before color was imagined to be possible), a few decades later we found ourselves locked into having the worst (most primitive) TV signals on earth.
- Interface standards also impose constraints on their consumers. An application that has been written to a well-supported standard can have high confidence of working on any compliant platform. But this warrantee is voided if the application uses any feature in any way that is not explicitly authorized by the standard. An application that uses non-standard features (or standard features in non-standard ways) is likely to encounter problems on new platforms or with new releases of the software providing those features. For many years, Windows users refused to upgrade to new releases because of the near certainty that a few of their devices and applications would stop working.
- When many different stake-holders depend on a standard, it often becomes difficult to evolve those standards to meet changing requirements:
    a. There will be many competing opinions about exactly what the new requirements should be and how to best address them.
    b. It seems that any change (no matter how beneficial) will adversely affect some stake-holders, and they will object to the improvements.

These trade-offs are fundamental to all standards. But there are additional considerations that make software interface standards particularly difficult.

Confusing interfaces with Implementations

Engineers are inclined to think in terms of design: how we will solve a particular problem? But interface standards should not specify design. Rather they should specify behavior, in as implementation-neutral a way as possible. But true implementation neutrality is very difficult, because existing implementations often provide the context against which we frame our problem statements. It is common to find that a standard developed ten years ago cannot reasonably be extended to embrace a new technology because it was written around a particular set of implementations.

The other way this problem comes about is when the interface specifications are written after the fact (or worse by reverse engineering):

- Since the implementation was not developed to or tested against the interface specification, it may not actually comply with it.
- In the absence of specifications, it may be difficult to determine whether particular behavior is a fundamental part of the interface or a peculiarity of the current implementation.

A common piece of Management 1A wisdom is:

> "If it isn't in writing, it doesn't exist"

This is particularly true of interface specifications.

The rate of evolution, for both technology and applications

The technologies that go into computers are evolving quickly (e.g. Wifi, Bluetooth, USB, power-management, small screen mobile devices), and our software must evolve to exploit them. The types of applications we develop are also evolving dramatically (e.g. from desk-top publishing to social networking). Nobody would expect to be able to pull the engine out of an old truck and drop-in replace it with a new hybrid electric motor. Yet everybody expects to be able to run the latest apps on their ancient phone, laptop or desktop for as long as it keeps running.

Maintaining stable interfaces in the face of fast and dramatic evolution is extremely difficult. Microsoft Windows was designed as a single-user Personal Computer operating system - making it very difficult to extend its OS service APIs to embrace the networked applications and large servers that dominate today's world. When faced with such change we find ourselves forced to choose between:

- Maintaining strict compatibility with old interfaces, and not supporting new applications and/or platforms. This is (ultimately) the path to obsolescence.
- Developing new interfaces that embrace new technologies and uses, but are incompatible with older interfaces and applications. This choice sacrifices existing customers for the chance to win new ones.
- A compromise that partially supports new technologies and applications, while remaining mostly compatible with old interfaces. This is the path most commonly taken, but it often proves to be both uncompetitive and incompatible.

There is a fundamental conflict between stable interfaces and embracing change.

Proprietary vs Open Standards

A *Proprietary* interface is one that is developed and controlled by a single organization (e.g. the Microsoft Windows APIs). An *Open Standard* is one that is developed and controlled by a consortium of providers and/or consumers (e.g. the IETF network protocols). Whenever a technology provider develops a new technology they must make a decision:

- Should they open their interface definitions to competitors, to achieve a better standard, more likely to be widely adopted? This costs of this decision are:

    - Reduced freedom to adjust interfaces to respond to conflicting requirements.
    - Giving up the competitive advantage that might come from being the first/only provider.
    - Being forced to re-engineer existing implementations to bring them into compliance with committee-adopted interfaces.
- Should they keep their interfaces proprietary, perhaps under patent protection, to maximize their competitive advantage? The costs of this decision are:
    - If a competing, open standard evolves, and ours is not clearly superior, it will eventually lose and our market position will suffer as a result.
    - Competing standards fragment the market and reduce adoption.
    - With no partners, we will have to shoulder the full costs of development and evangelization. In an open standards consortium, these costs would be distributed over many more participants.

This dilemma often complicates the development of interface standards. The participants may not only be trying to develop strong standards; Many are also trying to gain market position and protect prior investments.

Application Programming Interfaces

Most of the time, when we look up and exploit some service, we are working with Application Programming Interfaces (APIs). A typical API specification is *open(2)*, which includes:

- A list of included routines/methods, and their signatures (parameters, return types)
- A list of associated macros, data types, and data structures
- A discussion of the semantics of each operation, and how it should be used
- A discussion of all of the options, what each does, and how it should be used
- A discussion of return values and possible errors

The specifications may also include or refer to sample usage scenarios.

The most important thing to understand about API specifications is that they are written at the source programming level (e.g. C, C++ , Java, Python). They describe how source code should be written in order to exploit these features. API specifications are a basis for software portability, but applications must be recompiled for each platform:

- The benefit for applications developers is that an application written to a particular API should easily recompile and correctly execute on any platform that supports that API.
- The benefit for platform suppliers is that any application written to supported APIs should easily port to their platform.

But this promise can only be delivered if the API has been defined in a platform-independent way:

- An API that defined some type as int (implicitly assuming that to be at least 64 bits wide) might not work on a 32 bit machine.
- An API that accessed individual bytes within an int might not work on a big-endian machine.
- An API that assumed a particular feature implementation (e.g. *fork(2)*) might not be implementable on some platforms (e.g. Windows).

One of the many advantages of an Open Standard is that diverse participants will find and correct these problems.

Application Binary Interfaces

Well defined and widely supported Application Programming Interfaces are very good things, but they are not enough. If you are reading this, you are probably fully capable of re-compiling an application for a new

platform, given access to the sources. But most users do not have access to the sources for most of the software they use, and probably couldn't successfully compile it if they did. Most people just want to download a program and run it.

But (if we ignore interpreted languages like Java and Python) executable programs are compiled (and linkage edited) for a particular instruction set architecture and operating system. How is it possible to build a binary application that will (with high confidence) run on any Android phone, or any x86 Linux? This problem is not addressed by (source level) Application Programming Interfaces. This problem is addressed by Application Binary Interfaces (ABIs):

> An Application Binary Interface is the binding of an Application Programming Interface to an Instruction Set Architecture.

An API defines subroutines, what they do, and how to use them. An ABI describes the machine language instructions and conventions that are used (on a particular platform) to call routines. A typical ABI contains things like:

- The binary representation of key data types
- The instructions to call to and return from a subroutine
- Stack-frame structure and respective responsibilities of the caller and callee
- How parameters are passed and return values are exchanged
- Register conventions (which are used for what, which must be saved and restored)
- The analogous conventions for system calls, and signal deliveries
- The formats of load modules, shared objects, and dynamically loaded libraries

The portability benefits of an ABI are much greater than those for an API. If an application is written to a supported API, and compiled and linkage edited by ABI-compliant tools, the resulting binary load module should correctly run, unmodified, on any platform that supports that ABI. A program compiled for the x86 architecture on an Intel P6 running Ubuntu can reasonably be expected to execute correctly on an AMD processor running FreeBSD or (on a good day) even Windows. As long as the CPU and Operating System support that ABI, there should be no need to recompile a program to be able to run it on a different CPU, Operating System, distribution, or release. This much stronger portability to a much wider range of platforms makes it practical to build and distribute binary versions of applications to large and diverse markets (like PC or Android users).

Who actually uses the Application Binary Interfaces

Most programmers will never directly deal with an ABI, as it is primarily used by:

- the compiler, to generate code for procedure entry, exit, and calls
- the linkage editor, to create load modules
- the program loader, to read load modules into memory
- the operating system (and low level libraries) to process system calls

But it is occasionally used by programmers who need to write assembly language subroutines (for efficiency or privileged operations) that can be called from a higher level language (e.g. C or C++).